一、面向对象设计的原则SOLID

|  |  |  |
| --- | --- | --- |
| SRP | [单一职责原则](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk) | 一个类有且只有一个更改的原因。 |
| OCP | [开闭原则](https://docs.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/edit?hl=en) | 能够不更改类而扩展类的行为。 |
| LSP | [里氏替换原则](https://docs.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/edit?hl=en) | 派生类可以替换基类被使用。 |
| ISP | [接口隔离原则](https://docs.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/edit?hl=en) | 使用客户端特定的细粒度接口。 |
| DIP | [依赖反转原则](https://docs.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/edit?hl=en) | 依赖抽象而不是具体实现。 |

1. 单一职责原则

单一职责原则(Single Responsibility Principle，SRP）指出，一个类发生变化的原因不应该超过一个。这意味着代码中每个类，或者类似的结构只有一个功能。

在类中的一切都与该单一目的有关，即内聚性。这并不是说类只应该含有一个方法或属性。

类中可以包括很多成员，只要它们与单一的职责有关。当类改变的一个原因出现时，类的多个成员可能多需要修改。也可能多个类将需要更新。

下面的代码有多少职责？

class Employee {

  public Pay calculatePay() {...}

  public void save() {...}

  public String describeEmployee() {...}

}

正确答案是3个。

在一个类中混合了1)支付的计算逻辑，2)数据库逻辑，3)描述逻辑。如果你将多个职责结合在一个类中，可能很难实现修改一部分时不会破坏其他部分。混合职责也使这个类难以理解，测试，降低了内聚性。修改它的最简单方法是将这个类分割为三个不同的相互分离的类，每个类仅仅有一个职责：数据库访问，支付计算和描述。

1. “开-闭”原则

一个软件实体应当对外扩展开放，对修改关闭。对面向对象的设计过程来说，指的就是不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。

在Java语言中，可以设计出一个或多个抽象Java类或Java接口，规定出所有的具体类必须提供的方法的特征作为系统设计的抽象层。该抽象层在后面的实现过程中不再允许修改，即体现了“闭”的原则；同时，又可以为系统扩展出其他的可实现接口的具体类，体现了“开”的原则。

总结出这一原则一句话为：找到一个系统的可变因素，将它封装起来，并将这个命名为“对可变性的封装原则”。

1. 里氏替换原则

里氏替换原则(Liskov Substitution Principle,LSP)讲的是子类与基类之间的关系。适用于继承层次结构，指出设计类时客户端依赖的父类可以被子类替代，而客户端无须了解这个变化。

因此，所有的子类必须按照和他们父类相同方式操作。子类的特定功能可能不同，但是必须符合父类的预期行为。要成为真正的行为子类型，子类必须不仅要实现父类的方法和属性，也要符合其隐含行为。

一般来说，如果父类型的一个子类型做了一些父类型的客户没有预期的事情，那这就违反LSP。比如一个派生类抛出了父类没有抛出的异常，或者派生类有些不能预期的副作用。基本上派生类永远不应该比父类做更少的事情。

1. 接口隔离原则

接口隔离原则(Interface Segregation Principle，ISP)讲的是使用多个专门的接口比使用单一的总接口要好。一个接口应当简单地代表一个角色，而不是多个角色。

准确而恰当地划分角色以及角色所对应的接口，是面向对象设计中一个重要部分。将没有关系的接口合并在一起，形成一个臃肿的大接口，是对角色和接口的污染。

应用ISP时，类和他们的依赖使用紧密集中的接口通信，最大限度地减少了对未使用成员的依赖，并相应地降低耦合度。小接口更容易实现，提升了灵活性和重用的可能性。由于很少的类共享这些接口，为响应接口的变化而需要变化的类数量降低，增加了鲁棒性。

1. 依赖倒转原则(DIP)

依赖倒转原则(Dependence Inversion Principle，DIP)。高层次的抽象模块应该系统中复用的重点，也应该是关注的重点，那么低层次的具体实现应该依赖于抽象模块，而不是让抽象层次去依赖于具体的低层实现。这也就是依赖倒转原则，在实现的过程中，应该是依赖于上层的抽象层次而不应该是去依赖低层次的具体实现。

专业描述依赖倒转原则是:抽象不应该依赖于细节，细节应该依赖于抽象。也就是说要针对接口编程而不是针对实现编程。

针对接口编程就是说，应当使用java接口和抽象java类进行变量的类型声明、参量的类型声明、方法的返还类型声明，以及数据类型的转换等。

实现“开-闭”原则的关键是抽象化，并且从抽象化导出具体化实现。如果说“开-闭”原则是面向对象设计的目标的话，依赖倒转原则就是面向对象设计的主要机制。而里氏替换原则是依赖倒转原则的基础。

1. 迪米特法则

 Law of Demeter or Least Knowlegde Principle（LoD or LKP），迪米特法则或最少知识原则，这个原则首次在Demeter系统中得到正式运用，所以定义为迪米特法则。它讲的是“一个对象应当尽可能少的去了解其他对象”，一个对象只当知道它的直接合作者的接口。也就是又一个关于如何松耦合（Loosely-Coupled）的法则。